ENGR-UH 3511

Computer Organization and Architecture

Name: Nishant Aswani

Net ID: nsa325

Assignment Title: Lab 4

Microprocessor Design and Verilog HDL: Part 2

Nishant Aswani, nsa325@nyu.edu

Computer Organization and Architecture(ENGR-UH 3511), Instructor Cristoforos Vasilatos

1 Introduction

The MIPs CPU operates on the interconnection between several components. However, connecting these components is not as trivial as wiring them together. Often, because of the varied nature of the instructions, certain signals must be selected over others as inputs for a components. Hence, the CPU uses multiplexers (MUX) as well as control logic to select on these multiplexers.

The following lab uses Verilog to continue an implementation of a 32-bit MIPS CPU. The modules implemented include a program counter, instruction memory, register file, ALU, data memory, multiple MUXes, half-adders, sign-extension, and control logic

2 Methodology

A top-level module was generated through which the several CPU modules were connected. Output from a given module was connected to a wire in the top-level module. This wire was then used as an input for another module.

A testbench for the top-level moodule was used to test the CPU; the testbench was able to show the signals from all of the modules below.

3 Results

3.1 Assembly Program 1

The first assembly program consists of the following instructions

```
mem[0] <= 32'h00000000;     // EMPTY
mem[1] <= 32'h21290001;     //ADDI $t1 $t1 1
mem[2] <= 32'h214A0002;     //ADDI $t2 $t2 2
mem[3] <= 32'h21290001;     //ADDI $t1 $t1 1
mem[4] <= 32'h214A0002;     //ADDI $t2 $t2 2
```

The screenshot below shows the clocked program counter (PC) and instruction memory (IM). We see that there is a one cycle delay between each of the outputs. This is because the PC and IM are clocked.

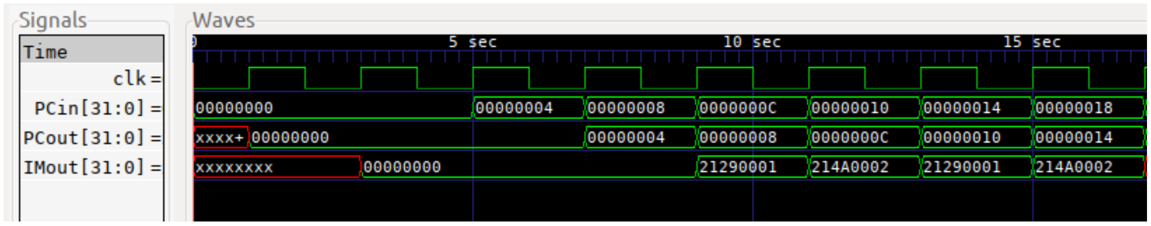

Figure 1: Wave output of program counter and instruction memory output

The instruction output is sliced, and there is a decision to be made for what must be fed into the register file. Hence, we have a mux that selects between `IMout[20:16]` and `IMout[15:11]` for the write register address. Below, we see that the mux selects for `IMout[20:16]` because that slice contains the target registers, such as `$t1` and `$t2`.

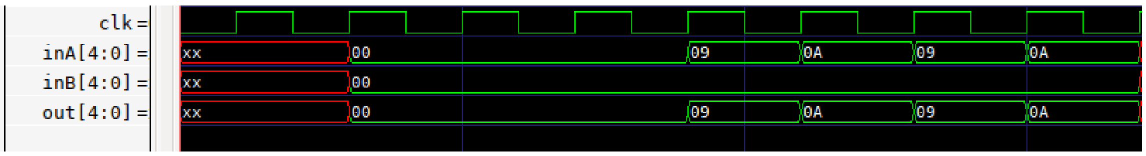

Figure 2: Wave output instruction memory mux

In parallel, the CPU must sign extend the 16 bits of the immediate value to 32 bits to be used in the ALU.

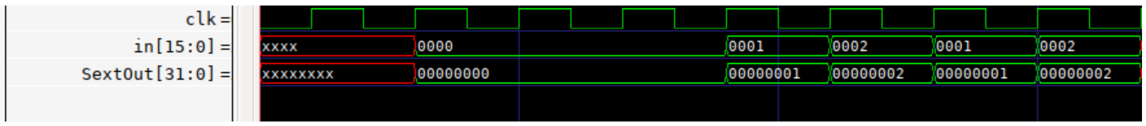

Figure 3: Wave output instruction memory mux

Finally, we send the sign extended immediate value and the output of readRegisterOne (which is readDataOne) into the ALU and we receive an output that is the sum of the two inputs.

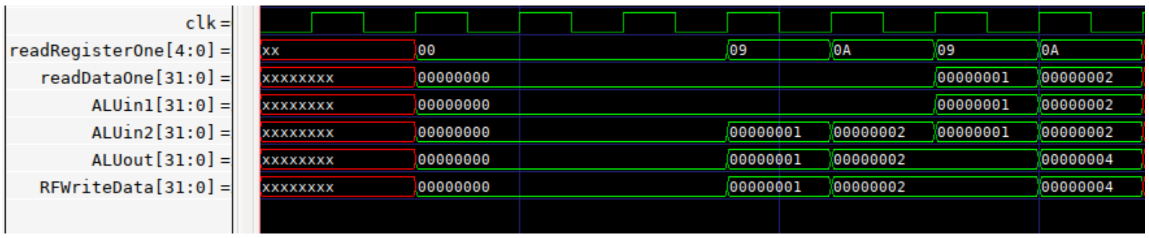

Figure 4: Wave output ALU

Notice that the `readDataOne` and `ALUin1` signals are identical. Also notice that the `ALUout` and `writeData` signals are identical. The latter is because the mux handling register writing selects the ALU output in this case, since it is not a load word instruction.

As expected, the writeData signal outputs: 1, 2, 2, and 4. Both registers `$t1` and `$t2` initially hold 0s. 1 is added to `$t1` twice and 2 is added to `$t2` twice. The output wave shows the values in the registers. testReg1 refers to `$t1` and testReg2 refers to `$t2`.

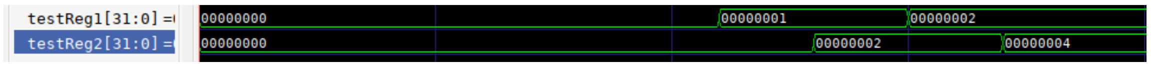

Figure 5: Wave output ALU

Thus, this program demonstrates the CPU's ability to fetch instructions from memory, read/write from a register, and compute something in the ALU.

3.2 Assembly Program 2

The second assembly program is as follows

```
mem[0] <= 32'h00000000;    // EMPTY
mem[1] <= 32'hAD2A0000;    //SW $t2 0($t1)
mem[2] <= 32'h8D290000;    //LW $t1 0($t1)
```

Because the data memory control signals vary between the two instructions, some additional control logic was added to the top-level module:

```
always@(*) begin
  // store word
  if(IMout[31:26] == 6'b101011) begin
    memWrite <= 1;
    memRead <= 0;
    RegWrite = 0;
  end

  // load word
  else if (IMout[31:26] == 6'b100011) begin
    memWrite <= 0;
    memRead <= 1;
    RegWrite = 1;
  end

  else begin
    memWrite <= 0;
    memRead <= 0;
  end
end
```

The logic above ensures that the `memRead, memWrite, and regWrite` signals function are as needed for the `sw` and `lw` instructions.

Once again, the screenshot below shows the clocked program counter (PC) and instruction memory (IM). We see that there is a one cycle delay between each of the outputs because the PC and IM are clocked.

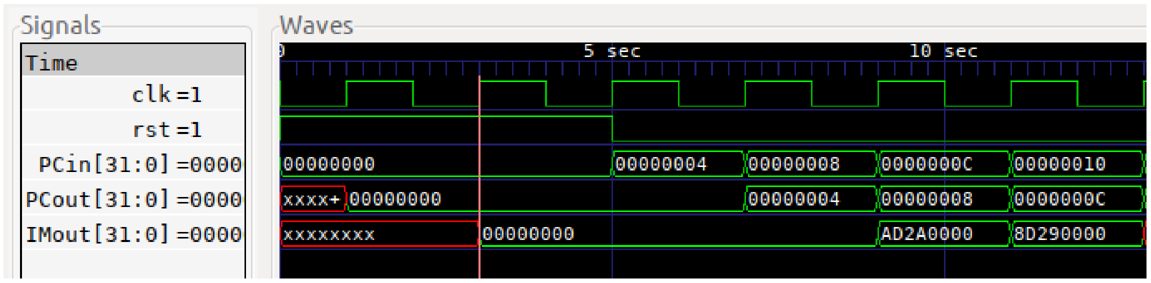

Figure 6: Wave output of program counter and instruction memory output

For this mini-program the `ALUsrc` is set to `1` so that the ALU accepts the sign-extended immediate value (which is the address offset) as its second argument. Because the offset in both the instructions is 0, we see that the `ALUin2` is `0x00000000` throughout.

To have a more meaningful output, the `$t1` register was hard-coded to have `0x00000001` as its value. This ”address” with an offset of 0, results in the ALU output sum `0x00000001`. So, the program will store to address `0x00000001` in the data memory for the first instruction. It will load from that address as well for the second instruction.

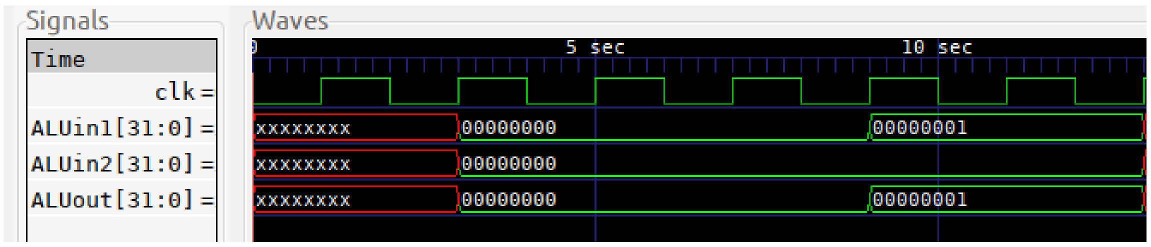

Figure 7: Wave output of the ALU

Below are the inputs and outputs of the data memory. Since the first instruction is a `sw` instruction, we see that the `memWrite` signal is briefly enabled. The next instruction is the `lw` instruction, hence the `memRead` signal is enabled.

The `address` signal refers to the data memory address. Before the first instruction is fetched, the address is default to `0x00000000`. However, as is shown in the figure above, register `$t1` stores `0x00000001`, which is used for both the instructions.

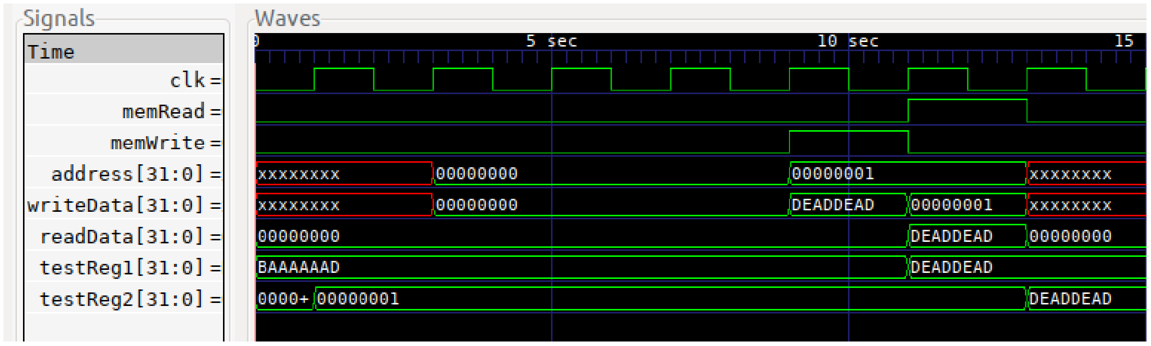

Figure 8: Wave output of the data memory and other relevant signals

Note that in the figure above, `testReg1` refers to the data memory address `0x0000001`, while `testReg2` refers to the register `&t1`.

Looking at the `writeData` signal, we see that register `$t2` holds `0xDEADDEAD`, which was stored into the data memory in the following clock cycle (see `testReg1`). The original value of `0xBAAAAAAD` was overwritten because of the `sw` instruction. Also, it is worth noting that the data memory intakes `0x00000001` as a `writeData` value, but this has no effect because `memWrite` signal is disabled by then.

The only change after `sw` has been:

```
dm.reg[1] = 0xBAAAAAAAD -> 0xDEADDEAD
```

On the next cycle, which is a `lw` instruction, the address in register `$t1` is offsetted by 0, resulting in `0x00000001`. So, the value in data memory at `0x00000001` is written into register `$t1`.

Therefore, we see the `readData` signal change from its default value of `0x00000000` to `0xDEADDEAD`, which is then reflected in the next clock cycle in register `$t1` (see `testReg2`).

Thus, after `lw`:

```
$t1 = 0x00000001 -> 0xDEADDEAD
```

This program demonstrates the CPU's ability to fetch from the instruction memory, read/write from a register, compute something in the ALU, and carry out store/load in the data memory.

4 Conclusion

As a result of connecting all components, several questions were raised about the clocking in the CPU. Certain components, such as the PC and IM, were able to implement the clock, as shown by the staggered output waves. However, other components, such as the ALU, could not be clocked as it would lead to accidentally overwriting the value of a different register. When initially attempting this lab, all components were clocked, leading to incorrect behavior. The clocking was then incrementally rolled back until the program function as desired. It is understood that if no components were clocked, this would be a single-cycle CPU.

Nevertheless, going about it this way helped explain the difference between `always @(*)` and `always @(posedge clk)` blocks in Verilog. The former block is sensitive to any changes on the RHS of the ”=” sign, and acts as the register version of the ”assign” keyword. On the other hand, the `always @(posedge clk)` block only updates the value on the LHS at the positive edge of the clock.

Working on this lab also clarified the difference between the ”=” and ”¡=” operators. The former acts as a sequential assignments, where each value is assigned one after the other. On the other hand, the second operator allows for parallel assignment.